IEEE DISTRIBUTED SYSTEMS ONLINE 1541-4922 © 2005 Published by the IEEE Computer Society
Vol. 6, No. 5; May 2005

A New Programming Model for Dependable Adaptive Real-Time Applications

Pedro Martins, *University of Lisboa*
Paulo Sousa, *University of Lisboa*
António Casimiro, *University of Lisboa*
Paulo Veríssimo, *University of Lisboa*

Editor's Note: The video clips in this article require Windows Media Player (http://www.microsoft.com/windows/windowsmedia/player/download/download.aspx) or RealPlayer (http://www.real.com/realsuperpass.html.

Aiming to reconcile uncertainty with an application's required predictability, this innovative programming model is based on fulfilling the coverage stability and no-contamination predicates. A video demonstration illustrates the model's practical value.

The convergence of several factors, including advances in wireless and sensory technologies, has created the conditions for constructing highly autonomous and proactive applications. The Information Society Technologies' CORTEX (*Co*operating *R*eal-time Sen*t*ient Objects: Architecture and *Ex*perimental Evaluation) project—under which we've developed, applied, and experimented with our work—considers just such applications.[1]

CORTEX envisions a new class of applications comprising (possibly many) smart components, called *sentient objects*, that can sense their environment and interact with it and each other by accepting and producing events. These interactions might occur in ways that demand predictable and sometimes guaranteed quality of service (QoS). However, achieving predictability is difficult given the objects' changing environments dictated by unstable and mobile object populations, unpredictable network load, varying connectivity, and failed system components.

We devised an innovative programming model that aims to reconcile uncertainty with an application's required predictability. The model is based on fulfilling two predicates—*coverage stability* and *no-contamination*—by programs or protocols subject to varying conditions of system

timeliness. You can use the model to construct applications that have several degrees of dependability. However, this requires understanding the effects of timing failures and, of course, being able to detect and recover from them. To do that, the CORTEX architecture relies on the existence of a timeliness wormhole called *Timely Computing Base* (TCB).[2]

In previous papers, we addressed the fundamentals underlying the development of dependable applications in wormhole-enabled systems.[3,4] In this article, we illustrate these concepts from a more practical point of view by discussing a video demonstration that we conceived in the context of the CORTEX project. (See the "**Key Concepts**" sidebar for more detailed definitions of the concepts we're discussing.)

Timely actions in the presence of uncertain timeliness

Today, we face a confluence of antagonistic aims when designing and deploying distributed systems. On one hand, our applications must achieve timeliness goals dictated both by QoS expectations regarding online services (such as time-bounded transactions) and by technical issues of real-time nature involved in deploying certain services (such as multimedia rendering). On the other hand, the open and large-scale environments in which applications and users execute and evolve exhibit uncertain timeliness or synchrony. Moreover, despite services' sometimes critical nature—whether related to money, privacy, or safety—they're more often deployed online or through open networks. They must be resilient to intrusions in the face of elusive attacks and pervasive and subtle vulnerabilities in relevant systems. In other words, the environment in which these services must operate exhibits uncertain behavior: we can't predict all possible present and future attacks; we can't diagnose all vulnerabilities.

So, what system model and design principles will let us meet uncertainty's grand challenge? We propose a novel design philosophy for distributed systems with uncertain or unknown attributes, such as synchrony, or failure modes. We built this philosophy on two principles:

1. Assume that uncertainty isn't ubiquitous or everlasting. This means that the system has some parts that are more predictable than others, and it tends to stabilize.

2. Be proactive in achieving predictability—that is, make predictability happen at the right time and place.

We can view these more predictable parts as shortcuts, or wormholes, through which it's possible to do things much faster or more reliably than apparently possible in the other parts of the system.

The wormhole concept can in fact be instantiated in different ways. For example, when applied in the security domain, a wormhole takes the form of a security kernel, a trusted component.[5] However, when timeliness is the relevant nonfunctional property to secure, the wormhole should be timely.

The TCB timeliness wormhole

Our approach to dealing with the timeliness requirements of sentient applications in environments of uncertain reliability and synchrony is based on the availability of a wormhole that provides timeliness guarantees. Paulo Veríssimo introduced the wormhole metaphor to illustrate one way of constructing distributed systems despite problems posed by the uncertainty of the operational environments.[6,7] Obtaining the properties of timeliness and trustworthiness must be done by *construction*, or by making some parts more predictable.

Because we're concerned with timeliness, we focus here on the TCB wormhole. The architecture of a system with a TCB would feature a *payload system* where protocols and application processes execute—more specifically, the "normal" system with several hosts interconnected by the Internet or intranet through which communication would take place, called the *payload network*. The architecture would also have a small alternative system, a "wormhole subsystem" or *control subsystem*, comprising local TCB modules interconnected by some form of medium, the *control channel*. Processes execute on several sites in the payload part, using the TCB by means of an appropriate TCB interface only when strictly needed.

The payload system can have any degree of synchronism. If bounds exist for processing or communication delays, their magnitude may be uncertain or unknown. Local clocks might not exist or might not have a bounded rate of drift toward real time. Components do only timing failures, including the failures' subsets omission and crash; no value failures occur. Conversely, the TCB subsystem (the wormhole) enjoys, by construction, a few synchronism properties: known upper bounds on processing delays, local clocks' drift rate, and message delivery delays. **Video 1** (.wmv (http://www.ieee.org/netstorage/computer_society/video01.wmv), .ram (http://www.ieee.org/netstorage/computer_society/video01.ram)) illustrates the TCB architecture.

In the TCB model, the TCB subsystem must cover synchrony assumptions. However, the TCB is a relatively small and simple part of the whole system. Consequently, enforcing strong synchronism properties for the TCB is much simpler than for the whole (payload) system. You can achieve much greater coverage of TCB properties than you can for the same properties in the payload system.

As a small part of the system, the TCB provides only three essential services: timely execution, duration measurement, and timing failure detection. The TCB provides these services to processes through the local TCB components, even when they have a distributed scope. More elaborate services, such as the QoS coverage service we'll discuss in a moment, are constructed outside the TCB but make use of these basic services.

Dependable applications in TCB-based systems

The availability of a TCB overcomes the negative effects of the payload system's lack of synchrony and reliability. This is understandable when you consider that the lack of timeliness lies, in the first

place, in the occurrence of timing failures. When timing failures occur, essentially three kinds of problems can arise: unexpected delays (the most immediate effect), decreased coverage, and contamination.[2]

Unexpected delays correspond to the violation of timeliness properties, which some applications might accept if they're prepared to work correctly under increased delay expectations. But a less intuitive effect is decreased coverage. When we make assumptions about the absence of timing failures, we have in mind a certain coverage, which is the correspondence between system timeliness assumptions and what the environment can guarantee. If the environment conditions start degrading to states worse than assumed, the probability of timing failure increases—that is, the coverage of those assumptions decreases. Another subtle consequence of timing failures is what we call the *contamination effect*, which occurs when a logical safety property of the system is violated due to timing failures.

To avoid these problems, ensure that coverage stays close to the assumed value, a condition expressed by the coverage stability property. Another solution is to ensure that timing failures' effect is confined to violating timeliness properties alone, as specified by the no-contamination property.

Unfortunately, not all applications can enjoy these properties and escape the consequences of uncertain synchrony. However, when assisted by a TCB, some applications classes can deal with timing failures' effects and achieve varying degrees of dependability. For example, the *fail-safe* class exhibits correct behavior or else stops in fail-safe state; the *time-elastic* class exhibits coverage stability; and the *time-safe* class exhibits no-contamination.

Our video demo, which we'll discuss in detail in a moment, shows applications of the fail-safe and time-elastic classes and the way in which they use the TCB to behave in a dependable way. We'll also see that applications of the fail-safe class use the TCB timing failure detection service to switch to a fail-safe state after the first failure. We'll observe how time-elastic applications can adapt essential timing variables to environment conditions (thus achieving coverage stability), using information indirectly provided by the TCB duration measurement service. In particular, we'll see how the TCB-based QoS coverage service[3] can be used to provide support for a dependable adaptation framework.

Construction of TCB wormholes

The feasibility of the TCB synchrony assumptions is an important issue that has already received our attention.[8] Here, we review the implementation we discussed in that paper, designed for communication over a wired infrastructure. Additionally, we describe a TCB implementation designed for wireless and mobile systems (used in our demo).

To secure the required synchrony properties, we must construct the TCB subsystem so that the rest of the system (the payload part) doesn't impair its timeliness. Furthermore, we must ensure the predictability of execution times within the TCB and communication delays over the control channel.

Concerning execution times' predictability, it's fundamental to use operating systems and hardware platforms with the necessary support for real-time operation. So far, we've implemented TCB prototypes for the most popular real-time versions of the Linux operating system (RTAI and RT-Linux), executing over a PC architecture, and also for the Windows CE operating system, executing over iPAQ Pocket PCs. These operating systems provide the necessary support in terms of priorities, scheduling, and other features. You can address potential predictability problems caused by PC hardware (such as loose control over shared buses) with specific solutions—for example, in the case of the PCI (peripheral component interconnect) bus.[9]

In currently available TCBs, the synchronous communication channel is based on a network that's physically different from the one supporting the payload channel and is used exclusively to connect local TCBs. The TCB model doesn't strictly require this dual network architecture, since in some networks it's possible to set up virtual channels with predictable timing characteristics, coexisting with essentially asynchronous channels.[10] Establishing a dedicated network for the control channel alleviated the problem of achieving predictability. In fact, since it's possible to control the network load, it's sufficient to find a feasible worst-case schedule given the available bandwidth.

We developed TCBs for both wired and wireless ad hoc networks. In the former case, we exploit some specific characteristics of switched Fast Ethernet networks (absence of collisions in micro-segmented topologies when operating in full-duplex mode) to enforce predictability at the media access control level. Message scheduling problems are automatically solved centrally at the switch.

Ensuring the timely operation of the TCB over wireless networks is a more complex task. In fact, the most commonly available technology for wireless LANs today (IEEE 802.11b) doesn't support real-time communication when operating in ad hoc modes.

Some proposals seek to endow ad hoc networks with real-time capabilities, such as the Time-Bounded Medium Access Control protocol[11] or the upcoming IEEE 802.11e standard,[12] which introduces the concept of traffic categories. However, since implementations of TBMAC nor 802.11e have not yet been made available, we developed a mock-up of a real-time MAC protocol for ad hoc wireless networks. Under certain conditions, this mock-up provides the desired properties and thus fully serves the purposes of our demo. More specifically, we use a simple token-based algorithm on top of a regular 802.11b channel to enforce a predictable medium-access latency, and we assume that it's possible to isolate the control channel from the payload channel using a dual network architecture, with two non-overlapping 802.11b ad hoc wireless networks.

Demonstration setup

Our demonstration shows how it's possible to build dependable adaptable applications that exhibit time- or coverage-elasticity and fail-safety. This demo illustrates the effect of timing failures, the importance of timely detection of timing failures, and the importance of using adaptation procedures. The demo is also intended to provide a more realistic picture of an application in which several mechanisms can be applied, so we use a scenario of cooperating cars that must behave according to a certain number of realistic rules.

Emulation system

Several classes of sentient applications have strict dependability requirements that emerge from safety rules that the environment imposes and that the system must preserve under any circumstance. The fulfillment of these requirements prevents incidents such as collisions between moving objects—for example planes and cars. In some of these applications, the computations must comply with timeliness constraints so that safety rules can be secured.

Testing such sentient applications in real scenarios using real hardware might not always be possible due to cost reasons or safety constraints. Therefore, we propose a software platform for emulating real environments, which can be a very useful and inexpensive tool to test subtle coordination and synchronization phenomena, difficult to reproduce or follow in real-life systems (see **Figure 1**).

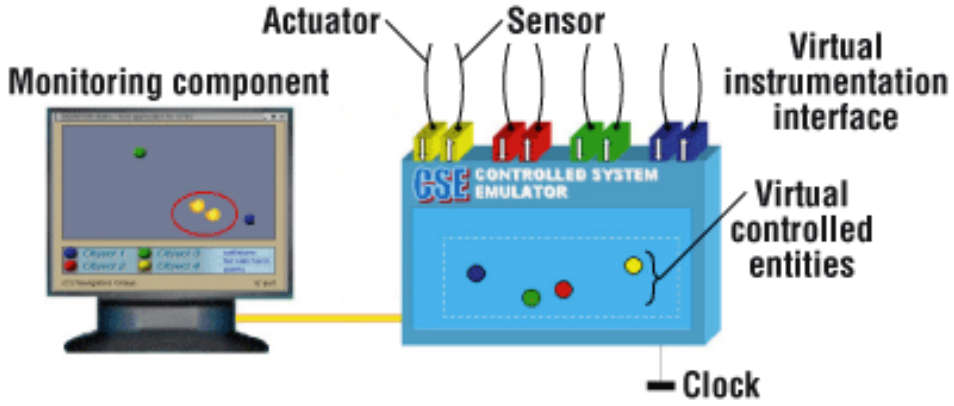

Figure 1. The emulation system lets us test coordination and synchronization phenomena.

The *Controlled System Emulator* is the software emulator's central component, or module, that emulates real environments. The environment emulated in the demo's first part, which we'll discuss in the next section, consists of a bi-dimensional space delimited by walls in which four virtually controlled entities move at a certain speed and in several directions. Four physical attributes define each entity: a position <*x*, *y*> in the plane, a shape, a speed, and a direction. For simplicity, but without loss of generality, we don't model physical laws such as friction or kinetic energy transfers in this environment. Therefore, in the absence of external control, entities tend to keep a constant speed and direction, unless they collide with a wall or another moving entity, in which case they will change direction.

The entities emulated inside the CSE are shaped as four colored balls: red, green, blue, and yellow. They can represent real objects with similar dynamics, such as cars, robots, or even subatomic particles. Initially, the balls start their movements with random speeds and directions.

The emulator's progress is triggered by a periodic clock signal with a certain predefined frequency (which affects our perception of the emulation speed but not of the emulation states). At each clock tick, the positions of all the controlled entities are updated according to their evolution rules, which are dictated by their speed and direction.

The emulation system has a monitoring component that provides a representation of the emulated environment. Periodically, the monitor gathers the state of the emulated environment from the CSE and updates its graphical representation. Note that this graphical representation doesn't have to be updated with the same frequency as that of emulator updates. In any case, it's guaranteed that the system will provide notification of all relevant events that occurred in the emulated environment (such as ball collisions).

Video 2 (.wmv (http://www.ieee.org/netstorage/computer_society/video02.wmv), .ram (http://www.ieee.org/netstorage/computer_society/video02.ram)) shows the emulation system's monitoring component in a scenario where the balls aren't being externally controlled by sentient objects. It's possible to observe the balls moving with a constant speed, following stable directions until they collide with a wall or with another ball, thus changing their direction. When ball collisions occur in the emulated environment, the CSE signals the monitoring component. The latter notifies human users about such collisions using special graphical representations (fireballs) and a collision counter displayed on the bottom right side of the window.

The CSE provides a virtual instrumentation interface comprising sensors and actuators to interact with the emulated environment. In our example, the CSE provides a sensor/actuator pair for each emulated ball. Through the sensor, a control application can know a ball's position, speed, and direction, and through the actuator, it can change an emulated ball's movement (speed and direction).

In distributed control applications (for example, if each ball is controlled from a different node), each sensor/actuator pair can be remotely accessed through a remote interface made available to each controller entity (each ball controller). To ensure bounded errors when sensing and actuating on the emulated environment from a remote interface, we must guarantee a timely connection between the remote instrumentation interface and the CSE central unit.

Sentient computing using emulated balls

We built an application comprising four sentient objects that control the movement of the emulated balls, trying to avoid collisions between them. This is the safety rule that must be ensured in this application.

Figure 2 shows the infrastructure that we used to build the emulation system and this application. The emulator runs in a dedicated machine that executes a real-time version of the Linux operating system (RTAI). The emulator CSE component runs as a hard real-time task of RTAI kernel and is executed periodically (this establishes the CSE's clock signal). The monitoring application is an X11 application (user-level process) that periodically acquires the state of the CSE and displays a

graphical representation of the emulated space and balls.

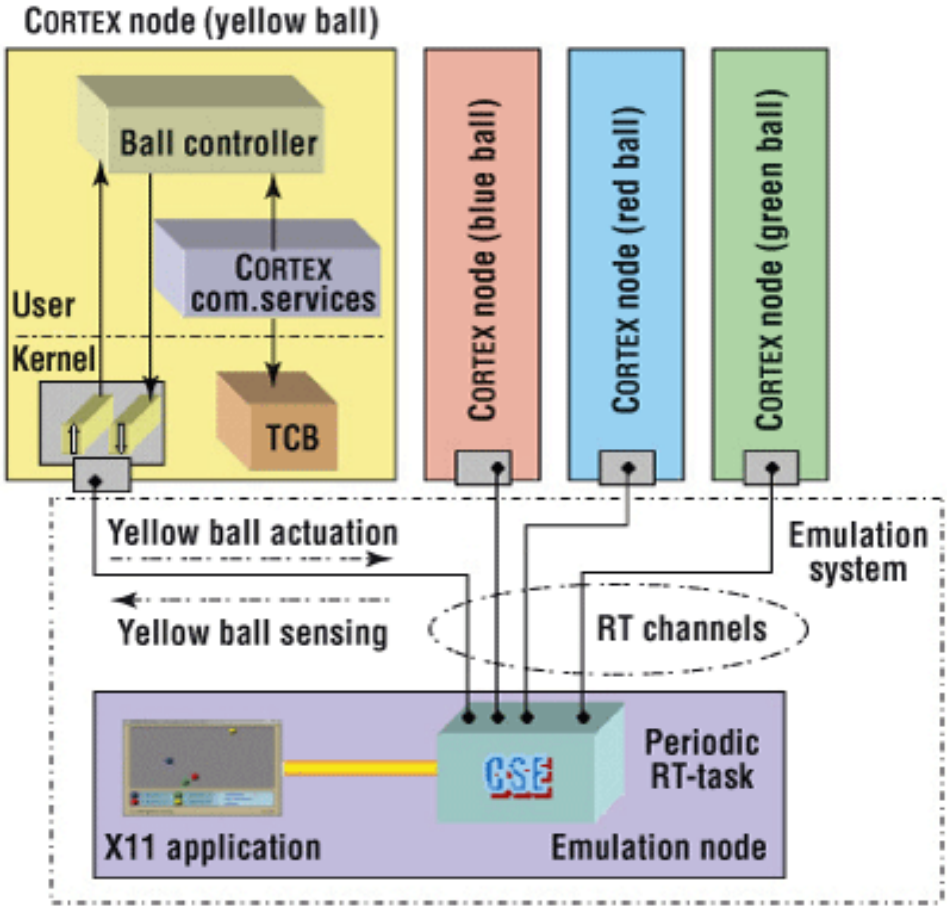

Figure 2. The emulation system infrastructure for the balls demonstration.

Each CORTEX machine runs a sentient object that controls one of the emulated balls. As explained previously, a remote instrumentation interface provides remote access. Dedicated real-time communication channels have been setup to allow a timely remote sensing and actuation.

Sentient objects communicate with each other through the payload channel. We use CORTEX communication services to publish and subscribe information concerned with the control algorithm. A sentient object publishes the position, speed, and direction of the ball it controls, and subscribes the same information from the other sentient objects. This lets every sentient object build a local image of the overall system. The control system will use this image to decide when and how to actuate on the controlled ball.

In this application scenario, it's fundamental to ensure that every ball controller has a consistent view (in time and space) of the environment, so that it takes the correct control decisions to avoid collisions. Therefore, the communication and processing needed to construct the local (real-time) image of the environment has to fulfill some timeliness requirements. For the control algorithm, it's sufficient to have a representation of each ball with a known and bounded positioning error. To

achieve that, we must define a maximum ball speed, and each controller must perform the following real-time computations:

- RTC_1: Publish the information of its controlled ball with a given period (to refresh other controllers' real-time images).
- RTC_2: Receive the information in a timely manner published by the other controllers (to update on time its real-time image). This also implies:
- RTC_3: Timely actuate in response to received information (for example, change the ball direction to avoid a possible collision).

The demo's objective is to show that despite the payload channel's lack of timeliness, we can use the TCB's *Timing Failure Detection* (TFD) service to secure the required safety property and avoid collisions. Furthermore, we'll see not only how we can use the feedback provided by the TCB's QoS coverage services (such as estimated delays for computations) to adapt the application behavior according to the changing environmental timing conditions, but also the utmost importance of being able to do it.

To observe the behavior of the application under several timeliness conditions, we have developed an additional demonstrator module (see **Figure 3**) that better controls the demo. It lets you

- inject artificial timing failures on the payload communication channel,
- enable or disable the use of TFD and/or QoS adaptation services,
- specify the desired level of coverage to be used in the QoS specification, and
- observe the delays estimated by the TCB for event dissemination.

Each ball controller is connected to this demonstrator module (the control panel) by a TCP/IP connection on the payload channel.

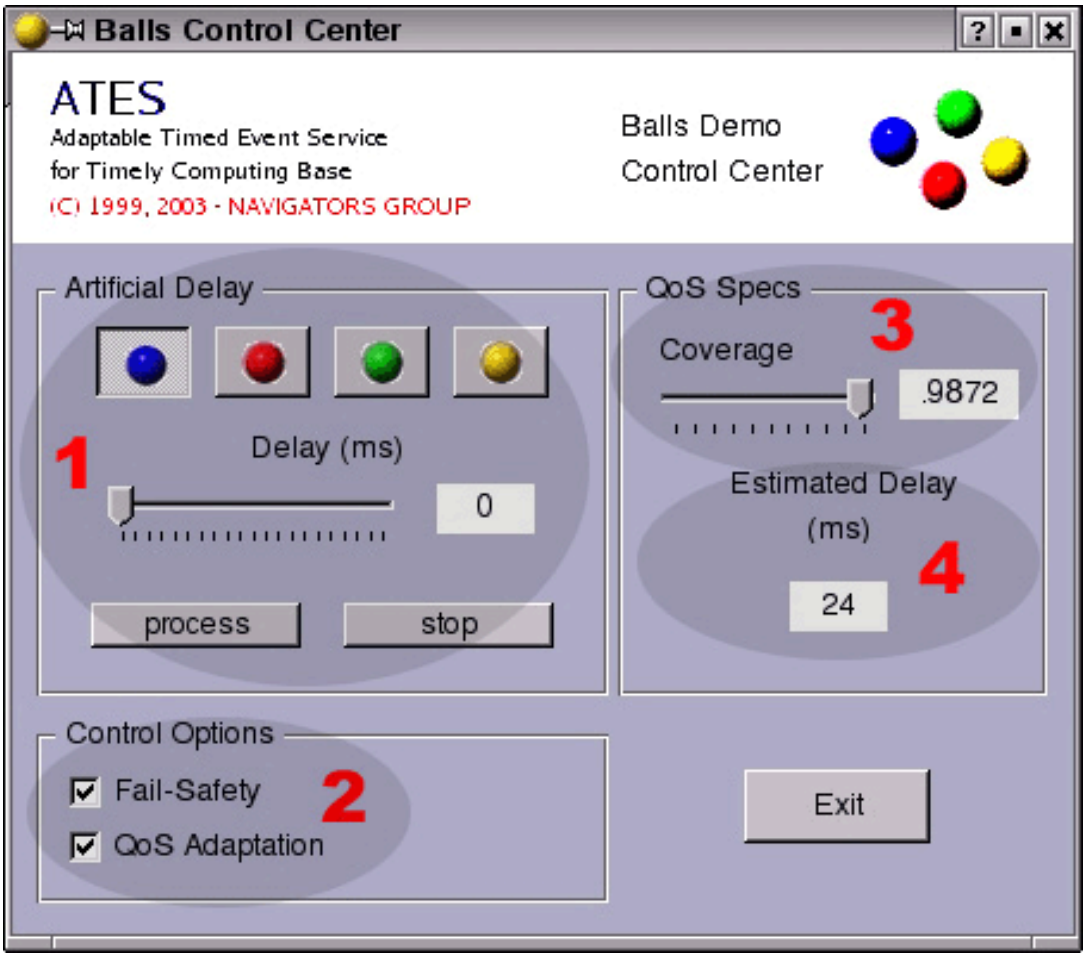

Figure 3. An additional demonstrator module control panel.

In the first part of our demo, we boot all the components of the balls demo setup (TCB, CORTEX communication services, emulator, ball controllers, and the control panel). The balls start immediately to be controlled, as shown in **Video 3** (.wmv (http://www.ieee.org/netstorage/computer_society/video03.wmv), .ram (http://www.ieee.org/netstorage/computer_society/video03.ram)). In this stage of the demo, the controllers use neither TFD nor QoS adaptation services.

Effect of timing failures. Highlighting the main subject of our demo, **Video 4** (.wmv (http://www.ieee.org/netstorage/computer_society/video04.wmv), .ram (http://www.ieee.org/netstorage/computer_society/video04.ram)) illustrates the nasty effects of timing failures in this particular sentient application. The video reproduces a demo sequence in which we start by injecting artificial delays of about 200 ms in the events disseminated by all the controllers (the use of TFD and QoS adaptation services is still disabled). Since QoS services aren't used, the controllers have no feedback from the TCB about the latency increase that consequently has resulted from the injected delays. Thus, from the controllers' knowledge of system timing variables, the dissemination latency of the events they receive remains within 10 ms (the estimated time bound before we have started to inject artificial delays), when in fact the real latency has increased to greater than 200 ms. This means that RTC_2 is no longer executed within the time bounds the control logic expects. Consequently, the real-time images kept internally by the

controllers that these timing failures are affecting (in this demo sequence, all of them) are not being updated on time. The control rule that assumes bounded positioning errors for the balls represented in the real-time images no longer holds—these images have become inconsistent. Furthermore, since the TFD service isn't used either in this part of the demo, the controllers have no way to be informed on time about the occurrence of timing failures. Therefore, they can't know when their real-time images aren't consistent, which, in such cases, might lead to incorrect decisions (actuations) and consequent ball collisions, as Video 4 shows.

Timely timing failure detection. Given that the application is of the fail-safe class (the fail-safe state corresponds to stopping a ball), every controller (sentient object) can use the TFD service the local TCB component provided to force the system to switch to a fail-safe state in a timely manner when a timing failure occurs—that is, when RTC_1, RTC_2, or RTC_3 aren't met.

Video 5 (.wmv (http://www.ieee.org/netstorage/computer_society/video05.wmv), .ram (http://www.ieee.org/netstorage/computer_society/video05.ram)) illustrates the importance of applications being able to quickly detect timing failures as a way to avoid ball collisions. The demo shows that a controller will promptly stop its ball whenever the TCB detects timing failures, and that it will recover as soon as its real-time image becomes consistent again. In the demo sequence reproduced in the video, we start by turning on the use of TFD in the control panel (QoS is still disabled in this part of the demo). Then, we inject artificial delays of about 106 ms in the events disseminated by the blue ball controller (only). As in Video 4, the controllers have no feedback from the TCB about latency changing. However, in this part of the demo, they use the TFD service, thus being able to force a timely fail-safe switch upon a timing failure detection. In the video, you can see the controllers (red, green, and yellow) stopping the balls in response to timing failures (generated by the injected delays) in the events received from the blue ball controller. We can observe also that the blue ball controller isn't affected by timing failures in the events it sends, since it gets its position, speed, and direction from its locally available virtual sensor, thus keeping its real-time image consistent. That's why the blue ball is still moving when the other ones have stopped. In this demo application, we don't inject timing failures in the RTC_1 and RTC_3 local computations. However, the application couples as well with these timing failures.

Adaptation of the QoS. Although using the TCB's TFD primitives ensures system safety, minimizing the activation of fail-safety procedures is also desirable. If possible, it's better to ensure that some progress is made, even if in a degraded operational mode (such as slowing down the speed of a ball). This keeps applications from being blocked on fail-safe states when their timing expectations about the infrastructure are unrealistically optimistic, such as in the scenario at the end of Video 5.

To adapt the essential variables while ensuring coverage stability (thus, dependably adapting), the controllers use the QoS services that the TCB provides to update their time bounds according to the available QoS.

The next demo sequence starts from the scenario at the end of Video 5, with the balls (red, green, and yellow) blocked on fail-safe states due to the artificial delays generated in the events disseminated by the blue ball controller. We start by enabling the use of QoS services in the control

panel. Since the system is using both TFD and QoS services, the controllers can couple with timing failures (as shown in Video 5), and furthermore they have feedback about the changing latency on the communication channel. This means they can adapt the time bounds stipulated for RTC_2 according to the delays estimated by the QoS service, minimizing the occurrence of timing failures and thus ensuring some application progress (keep the balls moving). Yet, it's still necessary to bound positioning errors of the balls represented in the real-time images, taking into account these longer delays. For that purpose, each controller adapts the speed of the ball it controls according to delays the TCB estimates. The general idea behind the control algorithm is to compensate the errors resulting from higher dissemination delays by reducing the speed of the balls.

After we have enabled the use of QoS adaption in the control panel, the controllers have started to receive the delays estimated by the TCB for event dissemination. In **Video 6** (.wmv (http://www.ieee.org/netstorage/computer_society/video06.wmv), .ram (http://www.ieee.org/netstorage/computer_society/video06.ram)), you can see in the control panel these delays increasing and also watch in the emulator's monitor the balls slowing down as a consequence of such higher delays. We follow our demo disabling again the use of QoS adaptation services and stopping all the artificial delays that we're injecting in the system. Afterward, we can observe the balls still moving slowly, not changing their speeds. This happens because QoS isn't used and hence the controllers had no feedback from the TCB about the new "good" environmental conditions that took place after we stopped all delay injections, which would have let the balls speed up. In such a case, the application isn't taking full advantage of the QoS provided by the communication channel. This issue reinforces the need for being able to dependably adapt to the infrastructure's changing timing conditions, and is illustrated clearly in the demo after we enable again the use of the QoS service. The speed of the balls starts increasing as a consequence of having again the controllers using the TCB's QoS services.

Finally, this part of the demo ends with a video in which we request several coverage levels to the QoS service. We can observe that higher levels of coverage lead to more pessimistic estimated delays and consequently to lower ball speeds. On the other hand, lower coverage requirements lead to more optimistic estimations (faster ball speeds) but at the cost of a higher probability for timing failure. This is reflected in the frequency of fail-safety activations, as **Video 7** (.wmv (http://www.ieee.org/netstorage/computer_society/video07.wmv), .ram (http://www.ieee.org/netstorage/computer_society/video07.ram)) shows. The applications must establish a suitable level of coverage, taking into account factors such as fail-over time, cost of a fail-safety activation versus benefits of having more optimistic estimated delays, and so on.

Video 8 (.wmv (http://www.ieee.org/netstorage/computer_society/video08.wmv), .ram (http://www.ieee.org/netstorage/computer_society/video08.ram)) shows a full demo sequence in which the controllers use TFD and QoS adaptation services.

A cooperating cars scenario

So far, we've demonstrated the fundamental mechanisms underlying the construction of adaptable real-time applications with the help of a TCB. However, we've enriched the demo with a more realistic setup, still showing how you can use an application's fail-safety and time-elasticity

characteristics to overcome the uncertainty of the environment in TCB-based systems.

This part of the demo was inspired by the vision of autonomous cars communicating and cooperating with other cars or entities in its proximity. The background scenario is a city area where only perpendicular streets exist. Three cars with different colors (red, green, or blue) are advancing on different streets and always in the same direction (up, down, left, or right). We modeled this virtual world as a small sphere, so that the cars are continuously meeting with each other at the same street crossings.

All cars have the same control logic, advancing if two conditions are satisfied: no car is in front, and no car is approaching from the right. This control logic is applied only to a safety area around the cars to avoid having one car stop in a crossing because another distant car is approaching from the right or the front. The car safety area depends on its speed—a higher speed results in a higher safety area.[3] **Video 9** (.wmv (http://www.ieee.org/netstorage/computer_society/video09.wmv), .ram (http://www.ieee.org/netstorage/computer_society/video09.ram)) shows the demo environment. The rectangle attached to each car is its safety area.

In this scenario, each car periodically disseminates an event with its position, speed, and direction, which every car can receive and use to construct a real-time image of the reality (an RT-image). These events must be timely delivered to achieve this RT-image and avoid car crashes. Intuitively, time bounds for event dissemination are related to the sender speed: a higher speed requires a smaller deadline. Because we're using a TCB-based system, a car can detect deadline violations of the events it is supposed to receive by using the TFD service. Then, it will promptly switch to a fail-safe state (for example, by stopping) because its RT-image might not be consistent with the reality. Moreover, each car can and should also adapt its speed to the environment conditions, following the same reasoning in the first part of the demo, thus ensuring coverage stability. An alternative mode of operation that we call coverage awareness is also shown. Here, a variable such as car speed is maintained even if the environment degrades, but the application is provided with the current lower coverage level. This mode intends to mimic practical situations where you want to privilege the value of a variable in detriment of its coverage, such as in an emergency, albeit in a controlled mode (risk-aware, in the car example).

In the demo, each car is represented by an iPAQ. The device's actual position is simulated in this demo, just because the iPAQs aren't equipped with a location mechanism such as the one provided by a GPS receiver. The speed is simulated and also changeable through the iPAQ's car interface (see **Figure 4a**). We've used the Windows CE TCB implementation to provide each iPAQ with a local TCB component. Therefore, iPAQs communicate using the 802.11b dual network architecture we described previously.

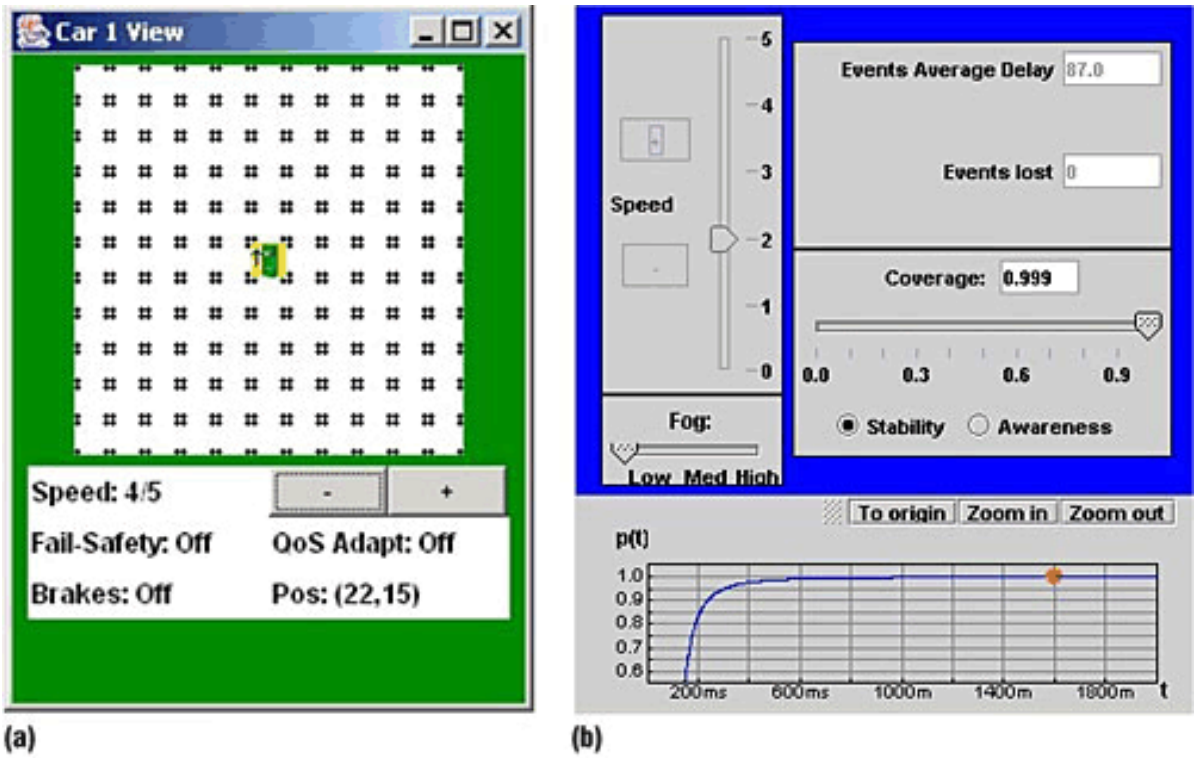

Figure 4. The iPAQ (a) car interface and (b) control panel.

Beyond the iPAQs, a monitor application is running in a laptop. **Figure 5** depicts the overall demonstrator architecture (the TCB control channel network is omitted, for clarity of presentation). **Figure 6** shows the iPAQs used in the demo. To implement the dual network architecture, each iPAQ is equipped with two wireless network cards: a Symbol Wireless Networker CompactFlash card and a Cisco Aironet 350 series PCMCIA (Personal Computer Memory Card International Association) card.

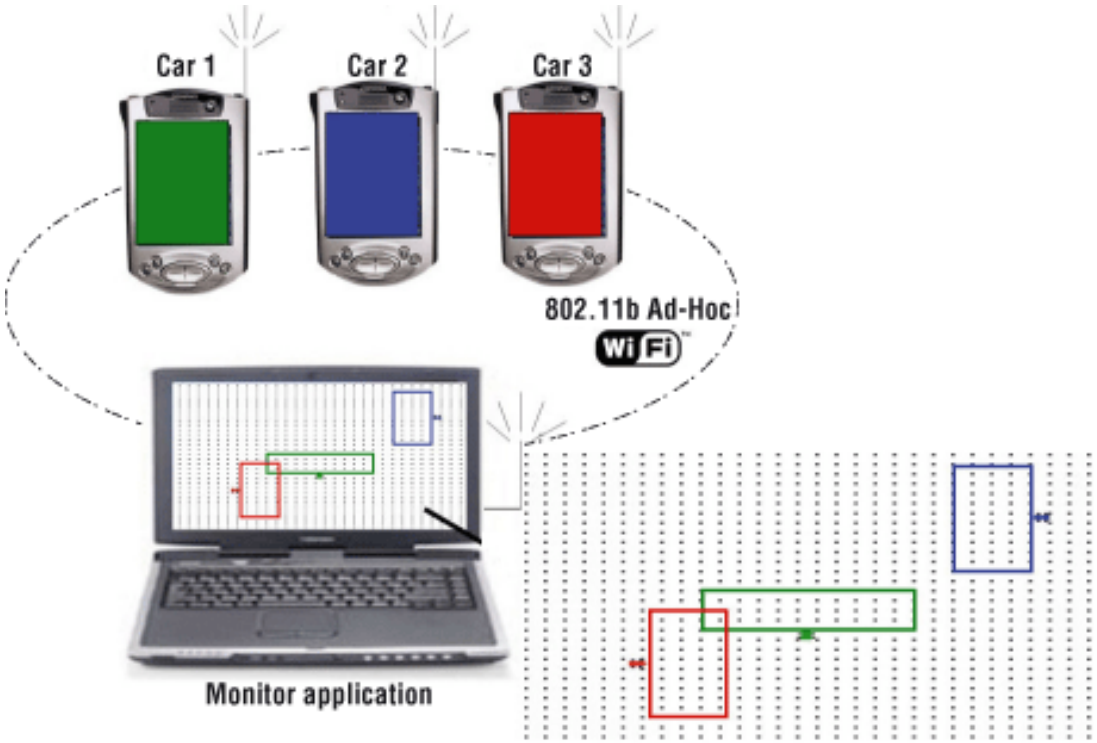

Figure 5. Cooperating cars demo architecture.

Figure 6. We used iPAQs to operate our cars.

The monitor application interface running on the laptop features three types of panels. A *Reality View panel* shows the actual reality, or an omniscient view, letting you observe each car's the behavior and detect car crashes (see **Figure 6**. Each of the three cars has a corresponding *Car*

Control panel (see **Figure 4b**) that lets you centrally control the car's speed and the parameters such as level of coverage of the QoS-adaptation feature. This panel also shows some car statistics, namely the average delay of received events and the number of lost events.

An interesting aspect of this demo is that when the QoS-Adaptation feature is enabled, you can continuously observe in the car control panels, the Probability Density Function generated by the TCB QoS coverage service to estimate the delays. This lets you observe how the environment conditions evolve and better understand the variations on speed (coverage stability) or on the coverage level (coverage awareness).

To observe the effects of fail-safety and QoS-adaptation, we can also control on the fly the cars' configuration—that is, whether they're using the fail-safety and QoS-adaptation mechanisms. You can do this in the *master control panel* of the monitor application.

In the iPAQ car interface, the top part of the screen displays the proximity view of the car—the car is always in the center because this is a relative view. In the bottom part of the screen, you can observe and change the car speed, observe if fail-safety and QoS-adaptation features are on or off, monitor the state of the breaking system (activated when some other car is nearby), and check the car's current position.

We divided the demo presentation into two major parts: the fail-safety demo and the QoS-adaptation demo.

Fail-safety demo. In this demo, an extra feature of the monitor application is the ability to inject artificial delays in the events a car receives. We call this artificial delay *fog*. You can select the injected fog's intensity from among three predefined values: low, medium, and high (see **Figure 4b**). The intensity directly determines which artificial delay to use. If we inject fog with a sufficient intensity while the fail-safety feature is turned off, car crashes are more likely to happen (see **Video 10**, .wmv (http://www.ieee.org/netstorage/computer_society/video10.wmv), .ram (http://www.ieee.org/netstorage/computer_society/video10.ram)). If we turn on the fail-safety feature, we see that cars stop before any car crash occurs, switching to a fail-safe state in response to a timing failure (see **Video 11**, .wmv (http://www.ieee.org/netstorage/computer_society/video11.wmv), .ram (http://www.ieee.org/netstorage/computer_society/video11.ram)). The fail-safe state is indicated with a siren that appears in the respective car control panel. A car recovers from a fail-safe state as soon as a timely event is received from each car from which a late event was received.

QoS-adaptation demo. Even without fog, when the fail-safety feature is on, cars obviously stop if they detect a timing failure. The probability of an event to suffer a timing failure is directly proportional to the sender speed. As mentioned before, a speedy car in a degraded environment could cause all other cars in its proximity (that must receive its events) to constantly stop because of timing failures. To avoid this, we turn on QoS-adaptation in coverage stability mode. If we select a sufficiently high level of coverage through the coverage slider presented in Figure 4b, cars adjust their speed to reflect the environment conditions and in this way the probability of a timing failure to occur decreases. **Video 12** (.wmv (http://www.ieee.org/netstorage/computer_society/video12.wmv), .ram

(http://www.ieee.org/netstorage/computer_society/video12.ram)) shows a degraded environment where cars are constantly stopping because fail-safety is on until QoS-adaptation is turned on in coverage stability mode with a desired coverage of 0.999.

If, for a certain car, we switch to coverage awareness mode, we use the QoS-adaptation feature to provide information about the level of coverage of the current speed, rather than to adapt the speed.

Conclusion

As future work, we plan to investigate the possibility of applying the proposed programming model to address well-known and fundamental problems, such as consensus or leader election. This would open new perspectives in terms of the way real-time systems are designed, based on the existence of fail-safe or time-elastic solutions for those fundamental problems.

Acknowledgments

This work was partially supported by the European Commission, through project IST-FET-2000 26031 (CORTEX); the FCT (Foundation for Science and Technology), through the Large-Scale Informatic Systems Laboratory (LaSIGE); and Microsoft Research, UK.

References

1. P. Veríssimo et al., "CORTEX : Towards Supporting Autonomousli and Cooperating Sentient Entities,"*Proc. European Wireless 2002* (EW 02), 2002, pp. 595–601.

2. P. Veríssimo and A. Casimiro, "The Timely Computing Base Model and Architecture," http://csdl.computer.org/comp/proceedings/dsn/2002/1597/00/15970027abs.htm, *IEEE Trans. Computers* , vol. 51, no. 8, IEEE CS Press, 2002, pp. 916–930.

3. A. Casimiro and P. Veríssimo, "Using the Timely Computing Base for Dependable QoS Adaptation," http://csdl.computer.org/comp/proceedings/srds/2001/1366/00/13660208abs.htm, *Proc. 20th IEEE Symp. Reliable Distributed Systems* (SRDS 01), IEEE CS Press, 2001,pp. 208–217.

4. A. Casimiro and P. Veríssimo, "Generic Timing Fault Tolerance Using a Timely Computing Base," http://csdl.computer.org/comp/proceedings/dsn/2002/1597/00/15970027abs.htm, *Proc. 2nd Int'l Conf. Dependable Systems and Networks* (DSN 02), IEEE CS Press, 2002, pp. 27–38.

5. M. Correia, P. Veríssimo, and N.F. Neves, "The Design of a COTS Real-Time Distributed Security Kernel,"*Proc. 4th European Dependable Computing Conference* (EDCC-4), LNCS

2485, Springer-Verlag, 2002, pp. 234–252.

6. P. Veríssimo, "Traveling through Wormholes: Meeting the Grand Challenge of Distributed Systems,"*Proc. Int'l Workshop Future Directions in Distributed Computing* (FuDiCo 02), 2002, pp. 144–151.

7. P. Veríssimo, "Uncertainty and Predictability: Can They Be Reconciled?"*Future Directions in Distributed Computing* , LNCS 2584, Springer-Verlag, 2003, pp. 108–113.

8. A. Casimiro, P. Martins, and P. Veríssimo , "How to Build a Timely Computing Base Using Real-Time Linux,"*Proc. 2000 IEEE Workshop Factory Communication Systems* (WFCS 2000), IEEE Press, 2000, pp. 127–134.

9. S. Schonberg, "Impact of PCI-Bus Load on Applications in a PC Architecture," http://csdl.computer.org/comp/proceedings/rtss/2003/2044/00/20440430abs.htm, *Proc. 24th IEEE Int'l Real-Time Systems Symp.* (RTSS 03), IEEE CS Press, 2003, pp. 430–439.

10. M. de Prycker, *Asynchronous Transfer Mode: Solution For Broadband ISDN*, 3rd ed., Prentice-Hall, 1995.

11. R. Cunningham and V. Cahill, "Time Bounded Medium Access Control for Ad-hoc Networks,"*Proc. 2nd ACM Int'l Workshop Principles of Mobile Computing* (POMC 02), ACM Press, 2002, pp. 1–8.

12. *Draft Supplement to Standard for Telecommunications and Information Exchange Between Systems–LAN/MAN Specific Requirements, Part 11: Wireless Medium Access Control (MAC) and Physical Layer (PHY) specifications: Medium Access Control (MAC) Enhancements for quality of service (QoS), IEEE Std. 802.11e/D3.0*, IEEE, May 2002.

Pedro Martins is a PhD student in computer science at the University of Lisboa. He's a member of the LaSIGE (Large-Scale Informatic Systems Laboratory) Navigators research group, participating in and contributing to national and international research projects such as DEAR-COTS (Distributed Embedded Architectures Using COTS Components), Micra (a model for Mission Critical Applications), CORTEX , and Maftia (Malicious and Accidental Fault Tolerance for Internet Applications). His research interests include fault-tolerant and real-time distributed systems, more specifically models and protocols for adaptive real-time systems. He received his masters degree in computer science from the University of Lisboa. Contact him at LaSIGE: Departamento de Informática, Universidade de Lisboa, Faculdade de Ciências, Bloco C6, Campo Grande, 1749–016 Lisboa, Portugal; pmartins@di.fc.ul.pt.

Paulo Sousa is a PhD student in computer science at the University of Lisboa. He's a member of the Navigators research group of LaSIGE. His research interests include the construction of secure distributed systems, namely the design of dependable intrusion-tolerant architectures and protocols for distributed systems. He received his BSc in computer science from the University of Lisboa.

He's a member of the ACM and the IEEE. Contact him at LaSIGE: Departamento de Informática, Universidade de Lisboa, Faculdade de Ciências, Bloco C6, Campo Grande, 1749–016 Lisboa, Portugal; pjsousa@di.fc.ul.pt.

António Casimiro is an assistant professor in the Department of Informatics, University of Lisboa Faculty of Sciences. He's a member of the Navigators research group of LaSIGE. He participated in and contributed to several national and international projects such as DINAS-DQS (Design and Implementation of CNMA-based Networks for CIME Applications in SMEs), CORTEX , and CaberNet. His research interests include fault-tolerant and real-time distributed systems, more specifically models and protocols for systems of partial synchrony and QoS-oriented and adaptive real-time systems. He received his PhD in informatics from the University of Lisboa. He is a member of the Ordem dos Engenheiros and the IEEE. Contact him at LaSIGE: Departamento de Informática, Universidade de Lisboa, Faculdade de Ciências, Bloco C6, Campo Grande, 1749–016 Lisboa, Portugal; casim@di.fc.ul.pt.

Paulo Veríssimo is a professor in the Department of Informatics, University of Lisboa Faculty of Sciences. He's also the leader of the Navigators research group of LaSIGE, a member of the European Security & Dependability Task Force Advisory Board, and an associate editor of the *IEEE Transactions on Dependable and Secure Computing*. He has coordinated the CORTEX project and was an Executive Board member of the CaberNet European Network of Excellence. His research interests include architecture, middleware, and protocols for distributed, pervasive and embedded systems, specifically the facets of adaptive real-time and fault/intrusion tolerance. Contact him at LaSIGE: Departamento de Informática, Universidade de Lisboa, Faculdade de Ciências, Bloco C6.3.10, Campo Grande, 1749–016 Lisboa, Portugal; pjv@di.fc.ul.pt.

Cite this article: Pedro Martins, Paulo Sousa, António Casimiro, and Paulo Veríssimo, "A New Programming Model for Dependable Adaptive Real-Time Applications," *IEEE Distributed Systems Online*, vol. 6, no. 5, 2005.

Key Concepts

Cortex

CORTEX (*Co*operating *R*eal-time Sen*t*ient Objects: Architecture and *Ex*perimental Evaluation) is an Information Society Technologies project whose main motivation was the belief that future mission-critical computer systems will consist of networked components that will react autonomously to myriad inputs to affect and control the surrounding environment. CORTEX 's key objective was to explore the fundamental theoretical and engineering issues necessary to support using sentient objects to construct large-scale proactive applications and thereby to validate using sentient objects as a viable approach to constructing such applications.

Sentient objects

Sentient objects accept input events from various sources (such as sensors), process them, and produce output events, actuating on the environment and/or interacting with other objects. Sentient objects can take several forms. They can simply be software-based components, or they can comprise mechanical and hardware parts—including the very sensor that substantiates "sentience"—mixed with software components.

Wormholes

Wormholes represent a new design philosophy for distributed systems. We argue that you can enhance a system with a more predictable part—the wormhole—through which some actions can be done faster and more predictably or reliably than apparently possible in the system's other parts. Applications/protocols run outside the wormhole but can use it for certain critical steps of their operation.

We borrowed the term *wormhole* from physics, where it designates a hypothetical topological feature of space-time that's essentially a shortcut from one point in the universe to another point in the universe.

Timely Computing Base wormhole

The TCB wormhole is a distributed embedded component that provides a set of time related services to client applications: timely execution of small functions, timing failure detection of timed executions, and duration measurement of local or distributed computations. You can use it as a fundamental building block for developing dependable real-time applications. The classes of applications that you can address with the help of a TCB include most soft or mission critical real-time applications, in particular fail-safe or quality of service (QoS) adaptive applications.

Dependable adaptation

Through dependable adaptation, applications/protocols can enjoy the property of coverage stability while preserving no-contamination. Dependable adaptation involves timely observation of the environment and rigorous mathematical analysis.

Coverage stability

Real-time applications/protocols are built under a set of timeliness assumptions about the environment they will operate in. Each of these assumptions has a certain probability of happening, which we designate as coverage. In an environment with uncertain timeliness, the coverage of a timeliness assumption naturally varies during system life. We say that an application/protocol benefits from the coverage stability property, if over an interval of mission the coverage of all its timeliness assumptions stays close to a certain assumed value.

No-contamination

An application/protocol typically comprises a set of safety properties, some of which are critical properties that can never be violated. Some noncritical safety properties might depend on timeliness assumptions. An application/protocol has no-contamination if none of its critical safety properties can be violated due to timing failures.

Fail-safe applications

An application of the fail-safe class exhibits correct behavior—that is, timely operation—or stops in a fail-safe state. Any class of application with a fail-safe state can be implemented using a TCB because the TCB can detect timing failures in a timely manner. You can use this ability to perform immediate shutdown to achieve fail-safety when the system becomes untimely—upon a timing failure that can't be handled, and before contamination occurs.

Time-elastic applications

Time-elastic applications are those whose time bounds can be increased or decreased dynamically, such as QoS-driven applications. The time-elastic class is oriented to securing coverage stability under a varying environment, for example achieving what we've called dependable QoS adaptation.

Time-safe applications

An application of the time-safe class is guaranteed to be free of contamination when timing failures occur, regardless of how it's implemented. Time-safe applications are those whose logical correctness doesn't depend on timeliness assumptions.